

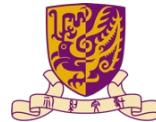
The 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)

Revealing Performance Issues in Server-side WebAssembly Runtimes via Differential Testing

Shuyao Jiang¹, Ruiying Zeng², Zihao Rao², Jiazen Gu¹, Yangfan Zhou², Michael R. Lyu¹

1. Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China

2. School of Computer Science, Fudan University, Shanghai, China

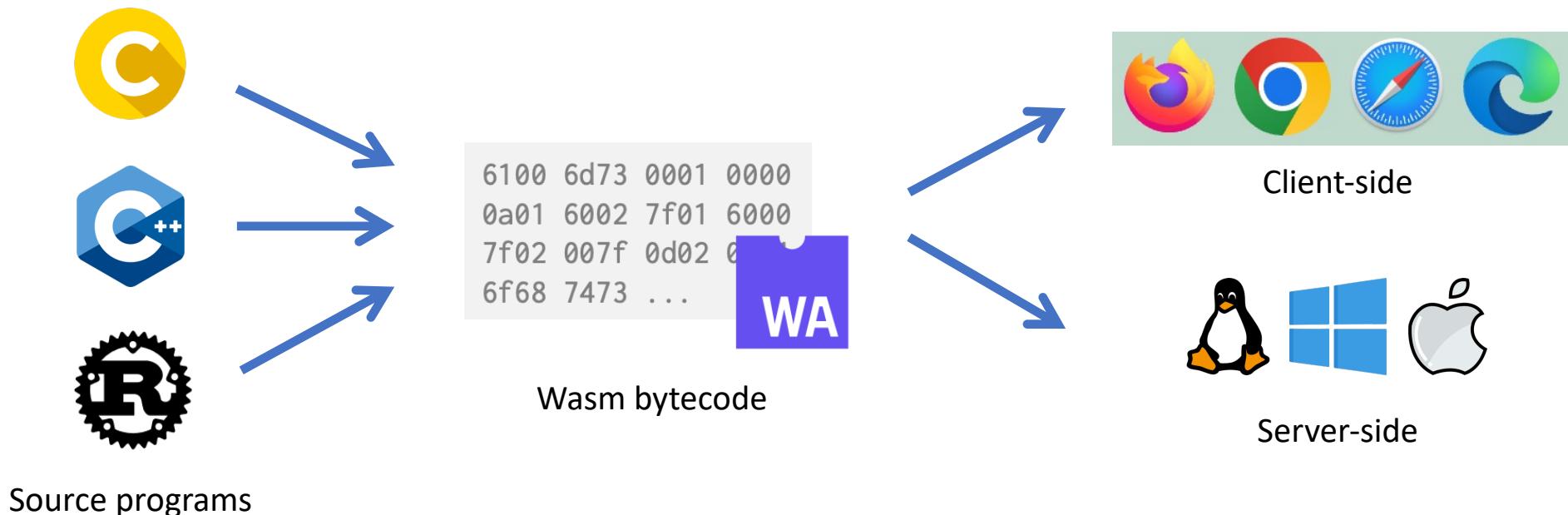


香港中文大學
The Chinese University of Hong Kong



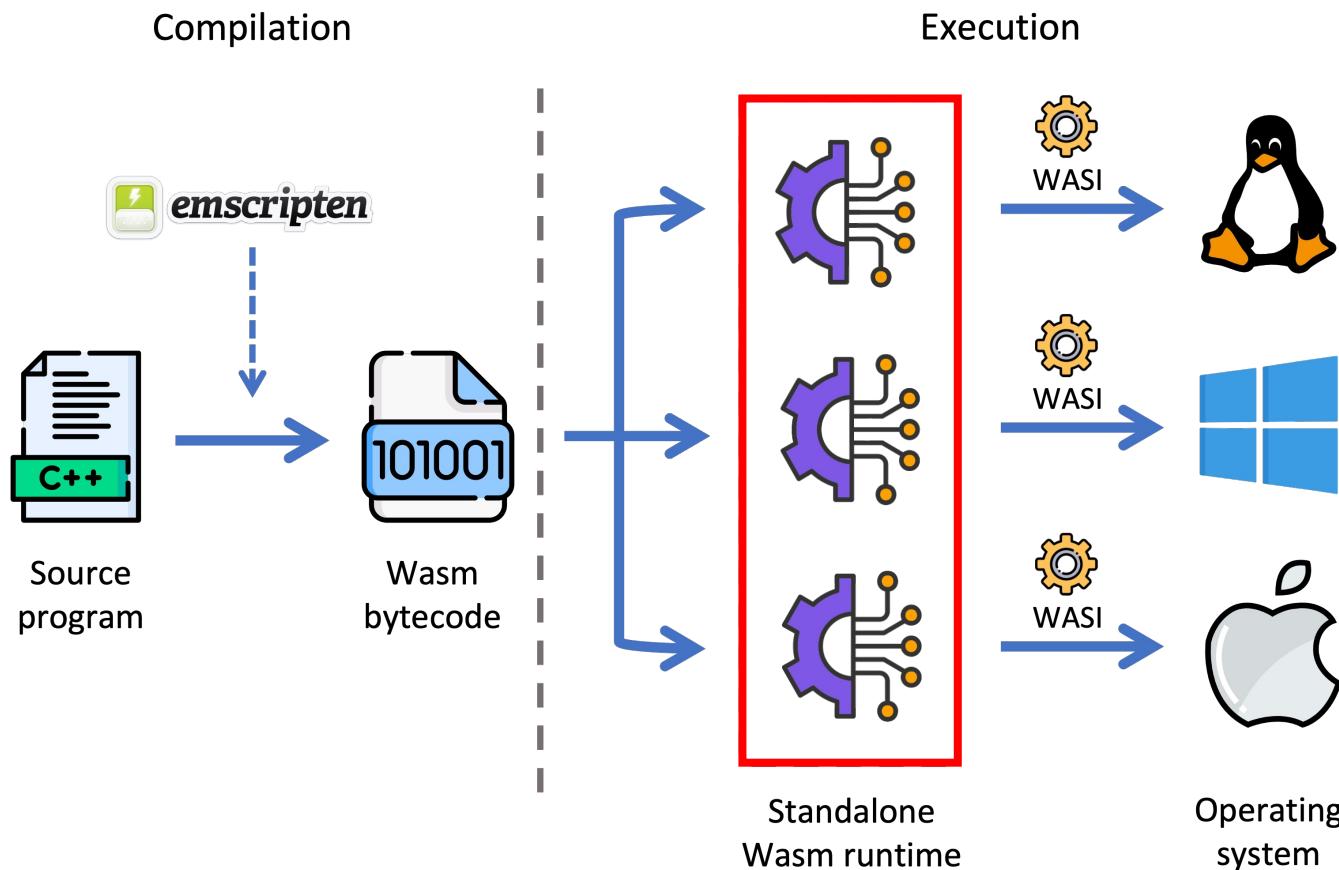
➤ WebAssembly (Wasm)

- A low-level bytecode format
- Fast, safe, portable
- Support in both browsers and server-side apps



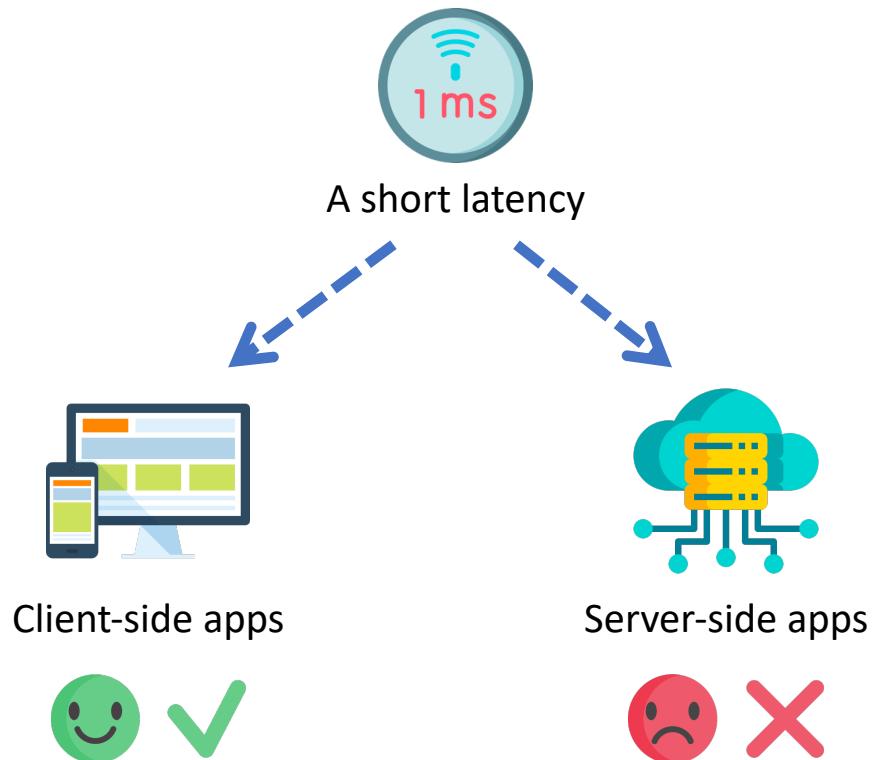
➤ Server-side Wasm Workflow

- Key component: **Standalone Wasm runtimes**



► Performance Issues in Server-side Wasm

- The impact of performance issues on the server side is usually greater than that on the client side.
- Standalone Wasm runtimes are still immature and more likely to cause performance issues.



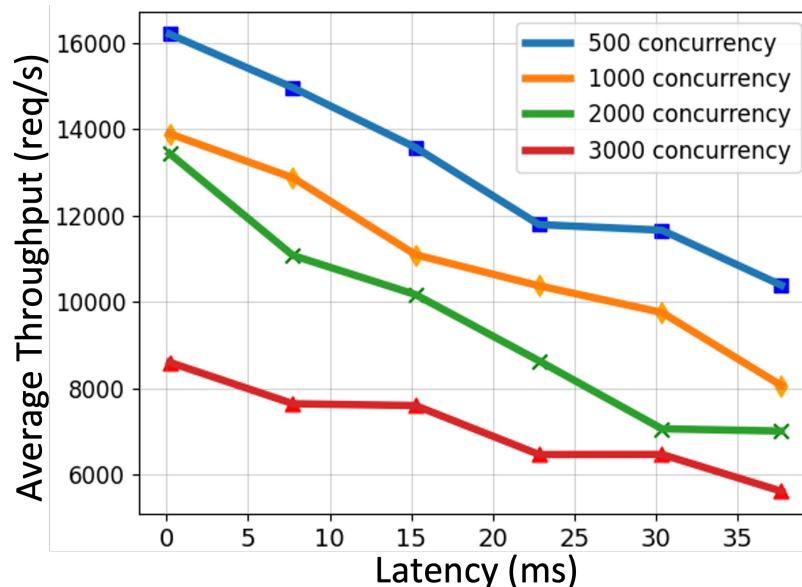
Major browsers: Well-developed



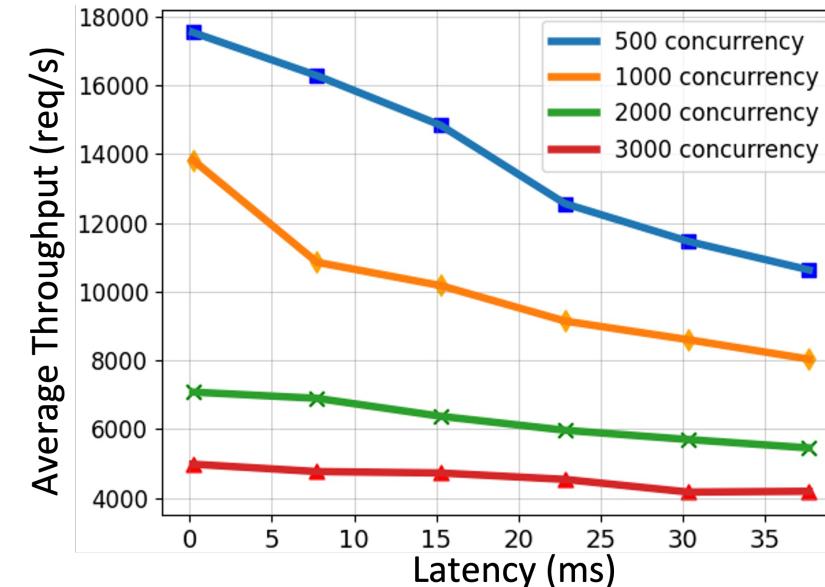
Standalone Wasm runtimes: Immature

Impact of Performance Issues: A Real Case

- Impact of WasmEdge runtime latency on service throughput
 - Service: [microservice-rust-mysql](#)



(a) 10,000 request



(a) 50,000 request

A 30ms-latency will result in a 20% to 50% drop in service throughput!

➤ Challenges & Solutions

- Our goal: Revealing performance issues in standalone Wasm runtimes



Challenge: Hard to manually analyze each Wasm runtime



Solution: Adopt the idea of **differential testing**



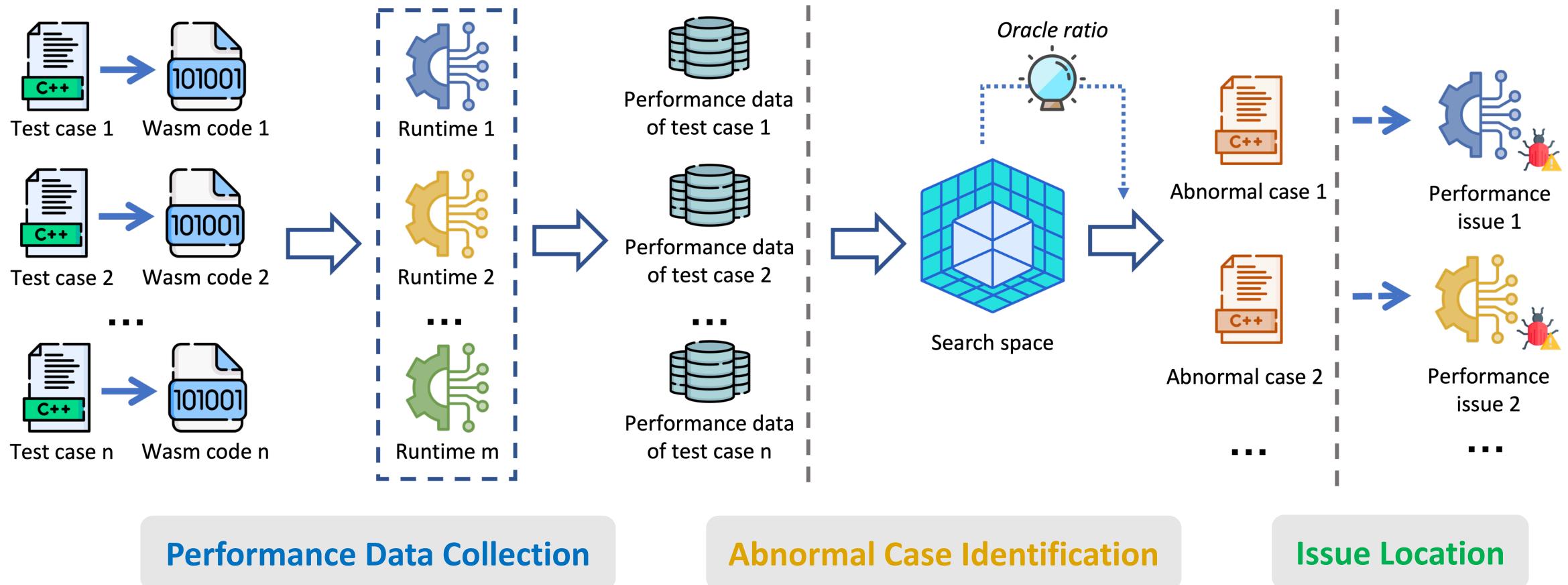
Challenge: Determine the oracle of performance issues



Solution: Propose an *oracle ratio* that reflects the systematic performance gaps among different Wasm runtimes

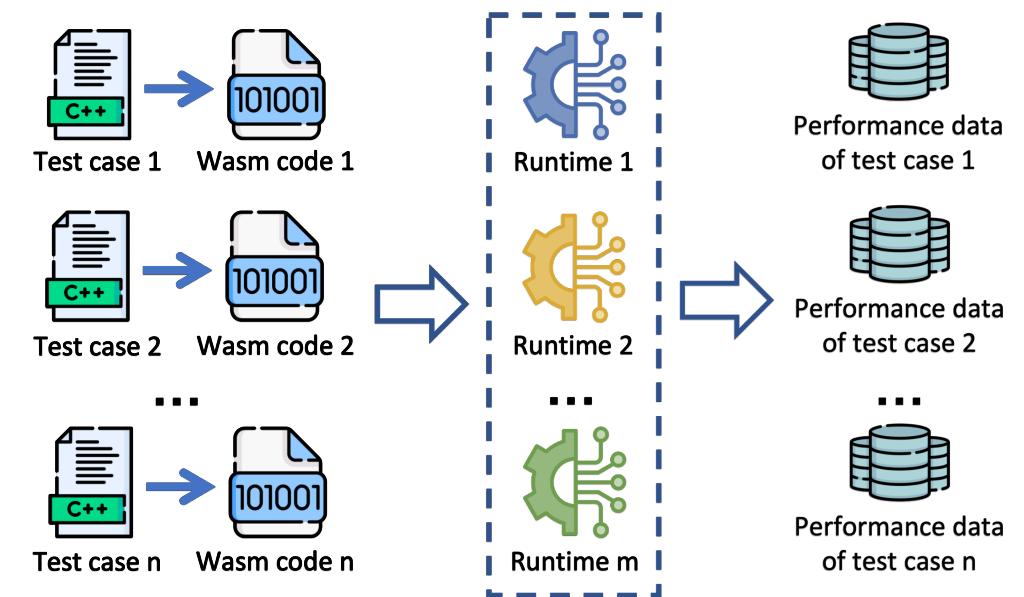
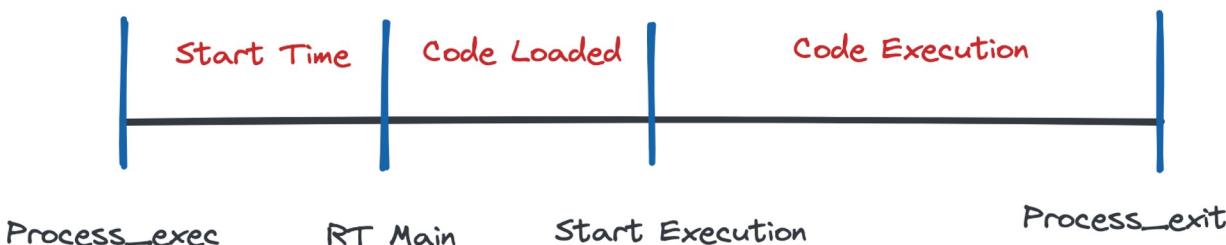
Approach: *WarpDiff*

- Wasm Runtime Performance Differential Testing



Phase 1: Performance Data Collection

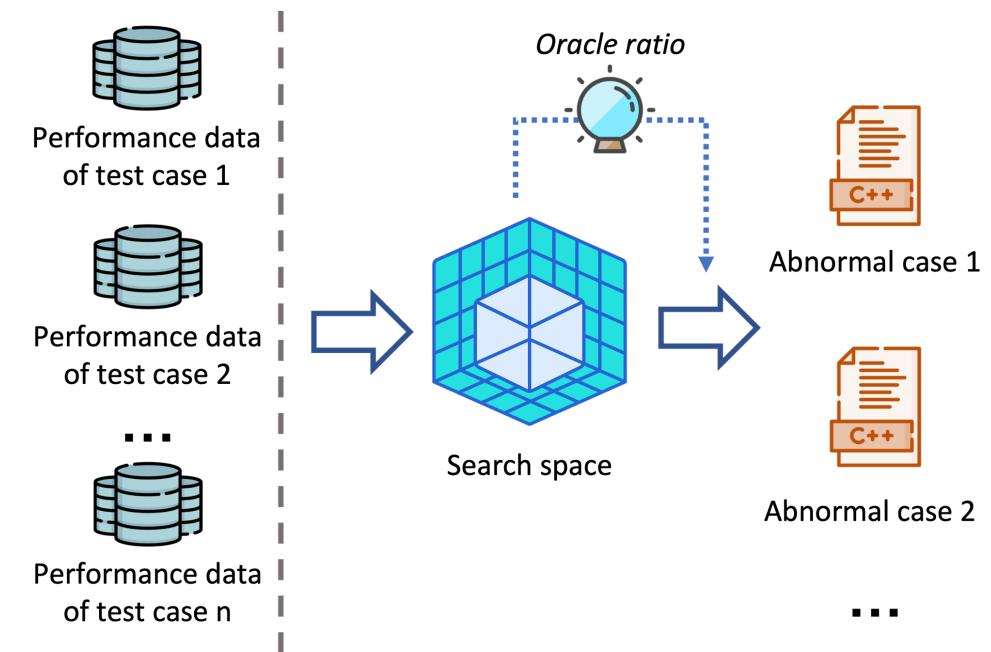
- Test case selection
 - Well supported by standalone Wasm runtimes
 - More likely to trigger performance issues
- Wasm code execution
 - Compile to Wasm → Execute on different runtimes
 - Ensure the correctness of the execution results
- Performance data recording
 - Three running stages



Performance Data Collection

Phase 2: Abnormal Case Identification

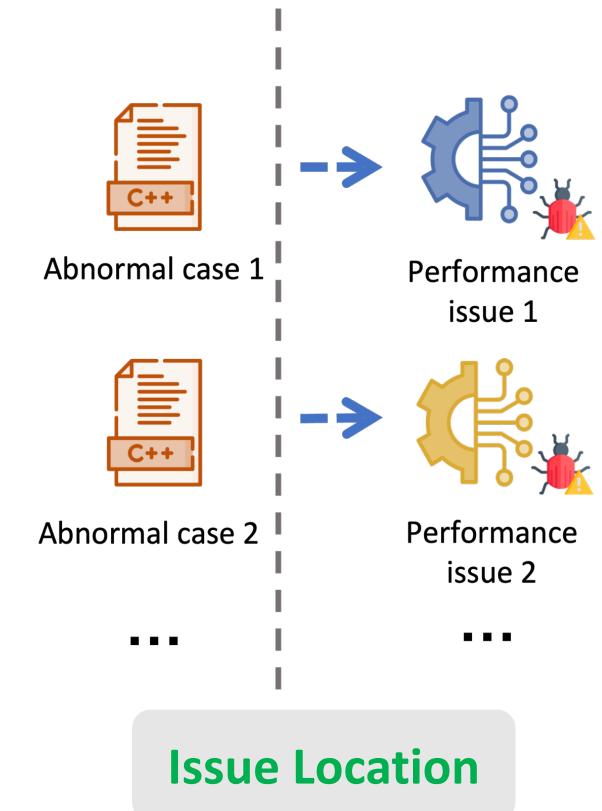
- **Key insight:** The execution time of the same test case on different Wasm runtimes should follow a stable ratio (*i.e.*, *oracle ratio*) in normal cases.
- How to represent the execution time ratio?
 - Vectorization for each test case
 - e.g., case x ran for 1s, 2s, 3s on three runtimes → the vector of x is $[1, 2, 3]$ → normalization
- How to determine the *oracle ratio*?
 - Take the center of all normalized vectors as the estimated *oracle ratio*
- Calculate the distance between a case vector and the estimated *oracle ratio*



Abnormal Case Identification

► Phase 3: Performance Issue Location

- **Goal:** Locate the runtime in which the performance issue occurs
- Analyze the impact of each runtime on the abnormal case
 - For each dimension in the case vector, adjust its value to make the case vector closest to the estimated *oracle ratio*
 - Record the adjustment value as *deviation degree*
- Treat the runtime with the largest *deviation degree* as the issue-related runtime



Research Questions

RQ1: How does *WarpDiff* perform in identifying performance issues in real-world standalone Wasm runtimes?

RQ2: What are the causes of the identified performance issues, and how can we verify them?

RQ3: What is the computational overhead of differential testing in *WarpDiff* ?

Experiment Settings

- Test cases
 - 141 C/C++ programs from *LLVM test suite*
 - Valid results on 123 programs
- Wasm runtimes for testing
 - Five Wasm runtimes with top *popularity* and *activity* on GitHub

TABLE I
INFORMATION OF OUR TEST CASES FROM THE LLVM TEST SUITE.

Benchmark	#Program	#LOC*	Benchmark	#Program	#LOC*
Adobe-C++	6	1,615	Misc-C++	7	1,322
BenchmarkGame	8	486	Misc-C++-EH	1	16,817
CoyoteBench	4	1,471	Polybench	30	4,364
Dhrystone	2	642	Shootout	14	573
Linpack	1	693	Shootout-C++	25	783
McGill	4	956	SmallPT	1	96
Misc	27	5,052	Stanford	11	1,135
Total		141			36,005

* LOC: lines of code.

TABLE II
INFORMATION OF WASM RUNTIMES FOR TESTING.

Runtime	#GitHub Stars*	Test Version	Execution Mode
Wasmer	15.1k	3.2.0	AOT
Wasmtime	12.1k	cli 8.0.0	AOT
Wasm3	6k	v0.5.0	Interpreter
WasmEdge	5.9k	0.12.0	AOT
WAMR	3.7k	1.1.2	Interpreter/AOT

* Statistics of Github stars is by April 2023.

RQ1: Identifying Performance Issues

- Top 10 abnormal cases
 - Based on the descending order of the *deviation degree* of the issue-related runtime

TABLE III
Deviation degree OF EACH RUNTIME SETTING ON THE TOP 10 ABNORMAL CASES.

Case	Wasmer	Wasmtime	Wasm3	Wasm3_compile	WasmEdge	WAMR	WAMR_AOT
BenchmarkGame/fasta.c	0.702	0.113	-0.248	-0.244	0.082	-0.270	0.081
Shootout/methcall.c	-0.051	-0.028	-0.164	-0.164	0.502	0.044	-0.014
Shootout-C++/methcall.cpp	-0.036	-0.031	-0.126	-0.128	0.415	0.072	-0.009
Shootout/random.c	0.075	0.315	-0.060	-0.060	0.079	-0.026	0.101
Shootout-C++/random.cpp	0.096	0.309	-0.063	-0.063	0.098	-0.036	0.121
Polybench/2mm.c	-0.038	-0.039	-0.151	-0.149	-0.035	0.268	0.003
Polybench/gemm.c	-0.038	-0.041	-0.145	-0.153	-0.036	0.267	0.007
Polybench/3mm.c	-0.037	-0.040	-0.145	-0.140	-0.034	0.261	0.005
Misc/flops-8.c	-0.019	0.012	-0.142	-0.142	-0.009	0.251	0.015
Misc/flops-4.c	0.234	-0.003	-0.127	-0.127	-0.019	0.168	0.001

Performance issues are common in existing standalone Wasm runtimes.

RQ2: Case Analysis

- Abnormal stage location → Fine-grained cause location → Cause verification

TABLE IV
SUMMARY OF PERFORMANCE ISSUES RELATED TO THE 10 ABNORMAL CASES.

Case	Related Runtime	Issue ID	Cause of Performance Issue	Status
BenchmarkGame/fasta.c	Wasmer	#3784	Improper implementation of <code>fd_write</code>	Confirmed
Misc/flops-4.c	Wasmer	#3821	Version issue of the <i>Cranelift</i> code generator	Confirmed
Shootout/methcall.c	WasmEdge	#2444	Improper handling when invoking function pointer	Confirmed
Shootout-C++/methcall.cpp	WasmEdge	#2442	Improper handling of virtual function	Confirmed
Shootout/random.c	Wasmtime	#6287	Insufficient optimization for division and modulo	Confirmed
Shootout-C++/random.cpp	Wasmtime			
Polybench/2mm.c	WAMR			
Polybench/gemm.c	WAMR	#2175	Insufficient optimization for matrix multiplications	Confirmed
Polybench/3mm.c	WAMR			
Misc/flops-8.c	WAMR	#2167	Insufficient optimization for complex arithmetic expressions	Confirmed

We summarize 7 performance issues for the 10 abnormal cases.

Case Analy

- Issue 1: Improper in

```

47 static void repeat_fasta (char
48     size_t pos = 0;
49     size_t len = strlen (s);
50     char *s2 = malloc (len + WI
51     memcpy (s2, s, len);
52     memcpy (s2 + len, s, WIDTH)
53     do {
54         size_t line = MIN(WIDTH,
55         fwrite (s2 + pos, 1, line,
56         putchar ('\n');
57         pos += line;
58         if (pos >= len) pos -= l
59         count -= line;
60     } while (count);
61     free (s2);
62 }
```

(a) Issue-related code snippet

- Issue 2: Version iss

Performance Issue in the fd_write Implementation #3784

[Open](#) hungryzzz opened this issue on Apr 19 · 7 comments [New issue](#)

hungryzzz commented on Apr 19 · Summary

Hi, I run the following case in different differences between wasmer and execute the wasm code(inner_module_run) than which in wasmtime .

- wasmer: 136486.78 us
- wasmtime: 30420.03 us
- wasmedge(AOT): 23816.45 us
- wamr(AOT): 20412.60 us

```
#include <stdio.h>
#include <sys/time.h>

typedef struct timeval timeval;
timeval tv;

static void repeat(int count)
{
    int len = 50;
    do {
        gettimeofday(&tv, NULL);
        count -= len;
        printf("advn", tv.tv_usec);
    } while (count >= 0);
}

int main()
{
    repeat(500000);
    return 0;
}
```

Hardware & OS

- Ubuntu 20.04
- CPU: Intel(R) Core(TM) i5-9500T CPU @ 2.20GHz
- Memory: 32GB

Emscripten

- emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 3.1.24 (68a9f990429e0bcfb63b1cde68bad792554350a5)
- clang version 16.0 (<https://github.com/llvm/llvm-project>) 277c382760bf9575ca2ec73d5ad1db91466d3f)
- Target: wasm32-unknown-emscripten
- Thread model: posix

Wasm runtime version

- wasmer: wasmer 3.2.0
- wasmtime: wasmtime-clif 8.0.0
- wasmedge: build from commit 381b7b28049b968297e6a585b92d1cba955def66
- wamr: iwasm 1.1.2

Additional details

I find that both wasmer and wasmtime use the cranelift as default compiler. I guess maybe it's related to the different version of cranelift. And then I try to use LLVM as the compiler (wasmer run --llvm), I get the execution time 423130.01us , which show more relation to the current cranelift in wasmer. So is it convenient to upgrade the version of current cranelift? Or how can I do it?

flops-4.txt

hungryzzz added the question label on Apr 25

pttSeb added this to the v4.0 milestone on Apr 25

pttSeb self-assigned this on Apr 25

Performance Issue related to Cranelift #3821

[Open](#) hungryzzz opened this issue on Apr 25 · 3 comments [New issue](#)

hungryzzz commented on Apr 25 · Summary

Hi, I run the attached case in different Wasm runtimes(after being compiled by Emscripten), and I also find some performance differences between wasmer and other 3 runtimes: the execution time(collected by perf-tool, probe begins when starting to execute the wasm code(inner_module_run in wasmer) and end in sched:sched_process_exit) in wasmer is 3.5x slower than which in wasmtime .

- wasmer: 2271270.05 us
- wasmtime: 610519.54 us
- wasmedge (AOT): 430803.42 us
- wamr (AOT): 418358.5 us

Hardware & OS

- Ubuntu 20.04
- CPU: Intel(R) Core(TM) i5-9500T CPU @ 2.20GHz
- Memory: 32GB

Emscripten

- emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 3.1.24 (68a9f990429e0bcfb63b1cde68bad792554350a5)
- clang version 16.0 (<https://github.com/llvm/llvm-project>) 277c382760bf9575ca2ec73d5ad1db91466d3f)
- Target: wasm32-unknown-emscripten
- Thread model: posix

Wasm runtime version

- wasmer: wasmer 3.2.0-alpha.1
- wasmtime: wasmtime-clif 8.0.0
- wasmedge: build from commit 381b7b28049b968297e6a585b92d1cba955def66
- wamr: iwasm 1.1.2

Additional details

I find that both wasmer and wasmtime use the cranelift as default compiler. I guess maybe it's related to the different version of cranelift. And then I try to use LLVM as the compiler (wasmer run --llvm), I get the execution time 423130.01us , which show more relation to the current cranelift in wasmer. So is it convenient to upgrade the version of current cranelift? Or how can I do it?

hungryzzz added the question label on Apr 25

pttSeb added this to the v4.0 milestone on Apr 25

pttSeb self-assigned this on Apr 25

Assignees
pttSeb

Labels
priority-medium ? question

Projects
None yet

Milestone
v4.x

Development
No branches or pull requests

Notifications
Customize
Subscribe
You're not receiving notifications from this thread.

4 participants

value of count
the value of count

with parameter 500000

ce Issue #3784.

➤ Case Analysis: WasmEdge

- Issue 3: Improper handling when invoking function pointer (#2444)
- Issue 4: Improper handling of virtual function (#2442)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct Toggle { // define a structure of Toggle
5     char state;
6     void (*activate) (struct Toggle);
7 } Toggle;
8
9 void toggle_activate(Toggle this) { // activate the toggle
10     this.state = !this.state;
11 }
12
13 int main() {
14     int i, n = 1000000;
15     Toggle tog;
16     tog.state = 1;
17     tog.activate = toggle_activate;
18
19     for (i=0; i<n; i++) {
20         tog.activate(tog); // invoke the function by pointer
21         // toggle_activate(tog); // invoke the function directly
22     }
23     puts(tog.state ? "true\n" : "false\n");
24     return 0;
25 }
```

Fig. 5. Simplified methcall.c related to Issue #2444 of WasmEdge.

➤ Case Analysis: Wasmtime

- Issue 5: Insufficient optimization for division and modulo (#6287)

```
16 inline double gen_random(double max) { // generate a random number
17     static long last = 42;
18
19     last = (last * IA + IC) % IM; // compound operations of *, + and %
20     return( max * last / IM ); // compound operations of * and /
21 }
```

(a) Issue-related code snippet of random.c.



```
1 #include <stdio.h>
2
3 int main() {
4     int N = 10000000, last = 42;
5     while (N--) {
6         last = (last + 33) % 13; // compound operations of + and %
7     }
8     printf("%d\n", last);
9     return(0);
10 }
```

(b) A new test case that can reproduce Issue #6287.

Case Analysis: WAMR

- Issue 6: Insufficient optimization for matrix multiplications (#2175)
- Issue 7: Insufficient optimization for complex arithmetic expressions (#2167)

```
90 #pragma scop
91 /* D := alpha*A*B*C + beta*D */
92 for (i = 0; i < _PB_NI; i++)
93     for (j = 0; j < _PB_NJ; j++)
94     {
95         tmp[i][j] = 0;
96         for (k = 0; k < _PB_NK; ++k)
97             tmp[i][j] += alpha * A[i][k] * B[k][j]; // alpha*A*B
98     }
99     for (i = 0; i < _PB_NI; i++)
100        for (j = 0; j < _PB_NL; j++)
101        {
102            D[i][j] *= beta; // beta*D
103            for (k = 0; k < _PB_NJ; ++k)
104                D[i][j] += tmp[i][k] * C[k][j]; // alpha*A*B*C + beta*D
105        }
106 #pragma endscop
```

Fig. 7. Issue-related code snippet of `2mm.c` in Issue #2175 of WAMR.

```
241 x = piref / ( three * (double)m ); //*****
242 s = 0.0; /* Loop 9. */
243 v = 0.0; //*****
244
245 for( i = 1 ; i <= m-1 ; i++ )
246 {
247     u = (double)i * x;
248     w = u * u;
249     v = w* (w* (w* (w* (w* (B6*w+B5)+B4)+B3)+B2)+B1)+one;
250     s = s + v*v*u*(((((A6*w+A5)*w+A4)*w+A3)*w+A2)*w+A1)*w+one);
251 }
```

Fig. 8. Issue-related code snippet of `flops-8.c` in Issue #2167 of WAMR.

RQ3: Computational Overhead

- Running time of the differential testing part in *WarpDiff*
 - With different numbers of runtime settings

TABLE V
COMPUTATIONAL OVERHEAD OF DIFFERENTIAL TESTING UNDER
DIFFERENT NUMBERS OF RUNTIME SETTINGS.

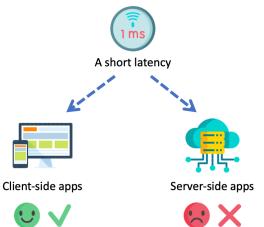
#Runtime	2	3	4	5	6	7
Avg. Overhead (s)	0.330	0.476	0.604	0.735	0.845	0.966
Std. Deviation	0.026	0.039	0.047	0.058	0.044	0.037

The computational overhead of differential testing only accounts for less than 0.01% of the whole process.

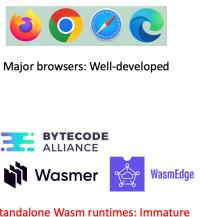
Conclusion

► Performance Issues in Server-side Wasm

- The impact of performance issues on the server side is usually greater than that on the client side.



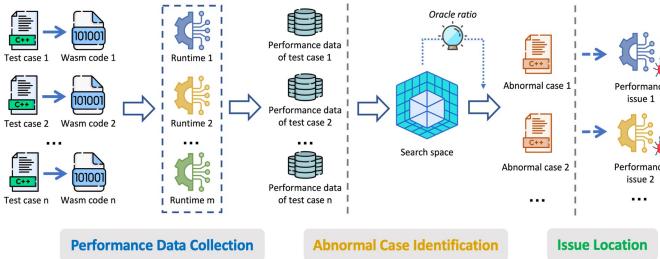
- Standalone Wasm runtimes are still immature and more likely to cause performance issues.



Significance

► Approach: WarpDiff

• Wasm Runtime Performance Differential Testing



Approach

► RQ2: Case Analysis

- Abnormal stage location → Fine-grained cause location → Cause verification

TABLE IV
SUMMARY OF PERFORMANCE ISSUES RELATED TO THE 10 ABNORMAL CASES.

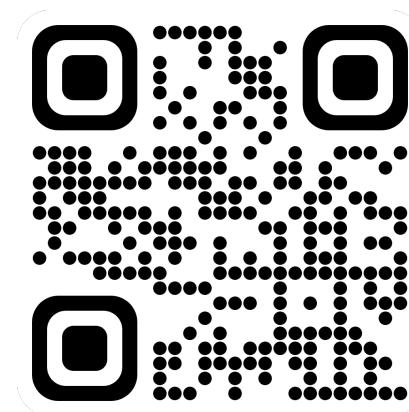
Case	Related Runtime	Issue ID	Cause of Performance Issue	Status
BenchmarkGame/fasta.c	Wasmer	#2784	Improper implementation of fd_write	Confirmed
Misc/flops-4.c	Wasmer	#3821	Version issue of the Cranelift code generator	Confirmed
Shootout/methcall.c	WasmEdge	#2444	Improper handling when invoking function pointer	Confirmed
Shootout-C++/methcall.cpp	WasmEdge	#2442	Improper handling of virtual function	Confirmed
Shootout/random.c	Wasmtime	#6287	Insufficient optimization for division and modulo	Confirmed
Shootout-C++/random.cpp	Wasmtime	#6287	Insufficient optimization for division and modulo	Confirmed
Polybench/2mm.c	WAMR	#2175	Insufficient optimization for matrix multiplications	Confirmed
Polybench/gemm.c	WAMR	#2175	Insufficient optimization for matrix multiplications	Confirmed
Polybench/3mm.c	WAMR	#2175	Insufficient optimization for matrix multiplications	Confirmed
Misc/flops-8.c	WAMR	#2167	Insufficient optimization for complex arithmetic expressions	Confirmed

We summarize 7 performance issues for the 10 abnormal cases.

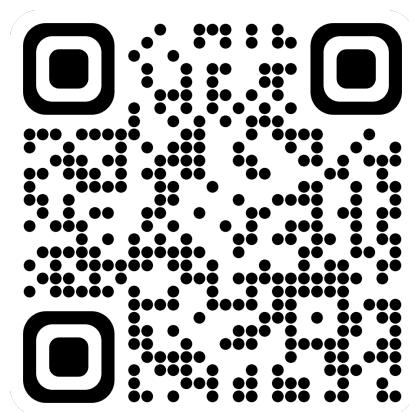
Results

Presenter: Shuyao Jiang

Email: syjiang21@cse.cuhk.edu.hk



Pre-print



Artifacts